Synchronization of PIM in iPhone and Evolution implementing a task application based on CouchDB

Author: Miguel Angel Rodelas Delgado
Tutor: Dr Andrew U. Frank

May 2010

INFORMACIÓN SOBRE EL PROYECTO FIN DE CARRERA REALIZADO EN ERASMUS

Autor	Miguel Ángel Rodelas Delgado
Tutor	Dr. Andrew U. Frank
Universidad	Technische Universität Wien
Coordinador Académico	Ascensión Gallardo Antolín
Cotutor en UCIIIM	Ascensión Gallardo Antolín
Fecha de lectura	31 de Mayo del 2010
Calificación obtenida	Matrícula de Honor

RESUMEN

Introducción

El aumento del número de dispositivos móviles en los últimos años, tales como portátiles, móviles y notebooks, han creado una nueva necesidad. El usuario quiere acceder a sus datos independientemente del lugar donde se encuentre: tanto en su ordenador personal en casa, con una conexión rápida y segura, como cuando se encuentra en el exterior, con conexiones débiles e intermitentes y de baja velocidad.

La conexión a una red no es siempre posible. Normalmente, hay problemas en las llamadas cuando se entra a un gran edicio o cuando se circula por las autovías. La conexión de datos es aún peor, siendo ésta en la actualidad aún bastante cara para el usuario medio. Debido a esto, el usuario preferirá descargar los datos una vez y sincronizar en vez de pagar por cada acceso.

En este contexto aparece el problema de la sincronización de datos. Es necesario mantener los dispositivos actualizados y operacionales incluso cuando no hay disponible una conexión. Los cambios hechos mientras que el dispositivo está desconectado tienen que ser transmitidos a los

demás dispositivos implicados tan pronto como se recupere la conexión.

La información a sincronizar podría incluir todos los posibles tipos de datos existentes en estos dispositivos, tales como fotografías, música o documentos. Sin embargo, este proyecto fin de carrera se centra en la sincronización de la información personal del usuario: contactos, calendarios, tareas, notas y correos electrónicos. Este conjunto de información personal de un usuario es normalmente conocido como PIM, de sus siglas en inglés Personal Information Management.

La sincronización de información personal conlleva una serie de dicultades propias de las características de este tipo de datos. Serán descritos estos retos mediante un sencillo ejemplo, una lista de contactos, en la sección 1.2. Entre otros, nos encontraremos el problema de cómo almacenar la información, cómo diferenciar los diferentes registros o cómo resolver conflictos cuando el mismo elemento ha sido modificado en diferentes dispositivos.

El objetivo de este proyecto fin de carrera es el de conseguir la sincronización de la información personal del usuario entre uno de los teléfonos móviles más vendidos actualmente, el iPhone de Apple, y una de las aplicaciones de escritorio más utilizadas en Linux, Evolution. Habrá que solventar bastantes problemas en el camino debido a que la API (Interfaz de programación de la aplicación, de las siglas en inglés: Application Programming Interface), y los servicios en general ofrecidos por Apple, siguen una política bastante cerrada y restrictiva. Dentro de las soluciones existentes, encontraremos que la sincronización no

es posible para las tareas de forma bidireccional, por lo que desarrollaremos nuestro propio gestor de tareas capaz de ser sincronizado entre el iPhone y Evolution. Para ello utilizaremos una nueva base de datos, CouchDB, que será explicada con mayor detalle en las siguientes secciones.

Estándares y protocolos

El proyecto fin de carrera comienza con un estudio detallado de los diferentes estándares y protocolos existentes para la sincronización de datos. En particular, son presentados los siguientes:

- SyncML (Synchronization Markup Language): protocolo de sincronización basado en mensajes XML (eXtensible Markup Language).
- WebDAV (Web-based Distributed Authoring and Versioning): conjunto de extensiones para HTTP (HyperText Transfer Protocol) que permiten la edición y gestión de ficheros en servidores web remotos.
- CalDAV (Calendar Distributed Authoring and Versioning): la extensión para calendarios, permitiendo el acceso a calendarios remotos.
  - iCalendar: éstandar existente para la representación de información relativa a calendarios.
- ActiveSync: protocolo de sincronización propietario desarrollado por Microsoft. Utilizado en la actualidad también por grandes empresas como Google o Apple que adquirieron la licencia para su uso.

- LDAP (Lightweight Directory Access Protocol): protocolo de aplicación para la consulta y modicación de datos usando servicios de directorios sobre TCP/IP.

Partes de la solución

A continuación se estudian las partes involucradas en la solución, detallando cuales son las opciones que ofrecen cada uno a la hora de sincronizar la información personal. Son principalmente tres:

- Teléfono móvil: la solución se centrará en iPhone, pero también serán comentadas las alternativas válidas para Android. Se diferenciarán entre soluciones nativas y aquellas que hacen uso de aplicaciones externas.
- Aplicación de escritorio que gestiona la información personal: Evolution. Igualmente, soporta algunos estándares, como WebDAV, de forma nativa, mientras que necesita el uso de programas externos para otros, como SyncML.
- Servidor para la sincronización de las partes anteriores: consideraremos tanto alternativas a instalar en nuestros propios servidores, como soluciones comerciales. Entre las posibilidades para un servidor propio, los candidatos tienen que soportar los estándares SyncML (para contactos) y CalDAV (para calendarios), al ser las únicas alternativas posibles para la sincronización de estos datos entre el teléfono y Evolution. Respecto a soluciones comerciales, Google es la

mejor alternativa en cuanto a facilidad de uso y eficiencia.

Hasta este punto sólo se ha hablado de protocolos para la comunicación de los cambios entre los diferentes dispositivos. También es importante analizar dónde y cómo es almacenada la información. Este proyecto compara y detalla las bases de datos relaciones, las bases de datos orientadas a objetos y los sistemas de control de versiones en relación a sus ventajas en cuanto a un sistema de sincronización. Finalmente, CouchDB, una base de datos orientada a objetos es elegida, debido a sus buenas características para la replicación y la gestión de conflictos, procesos primordiales en la sincronización de datos, más importantes que la estructuración de los datos en sí.

Solución propuesta y sus limitaciones

La parte más difícil de un proceso de sincronización es la detección y resolución de conflictos. Un conflicto ocurre cuando el mismo elemento ha sido modificado en diferentes dispositivos. Al sincronizar, hay que detectar este conflicto y resolverlo de forma que no se pierda información o que ésta sea errónea.

En la sincronización de información personal, la situación ideal es la fusión de los datos siempre que esta operación sea compatible. Por ejemplo, si en el dispositivo A se añade un nuevo campo de correo electrónico y en el dispositivo B se añade un nuevo teléfono para un mismo contacto, al sincronizar ambos dispositivos deberían de

disponer de ambos campos en ese contacto. Cualquier otra solución sería subóptima.

El mayor problema aparece cuando esta fusión no es posible. Por ejemplo, el dispositivo A modifica un número de teléfono de un contacto y el dispositivo B hace lo mismo 5 minutos después. Para resolver este tipo de conflictos, la solución más lógica es seguir la política "El más reciente gana", ya que es la información más actualizada de las dos opciones disponibles. Para poder actuar de este modo necesitaríamos una etiqueta de tiempo que nos permita en el servidor comparar y elegir el más reciente.

Ésta sería la solución ideal para la sincronización de información personal. Sin embargo, no es posible llevarla a cabo en la práctica ya que ninguna de las tres partes implicadas cumple los requisitos necesarios. Por una parte, los clientes no añaden los campos necesarios (la etiqueta de tiempo y el identificador único). Por otra parte, los servidores no soportan la política "El más reciente gana" y la fusión está normalmente bastante limitada.

Implementación de la solución

Como se ha comentado anteriormente, la única posibilidad para sincronizar el iPhone y Evolution mediante nuestro propio servidor y con alternativas de código abierto, es el uso de los estándares SyncML para contactos y CalDAV para calendarios. En el servidor se pueden utilizar opciones que soportan ambos protocolos, como eGroupware, o servidores dedicados, como Funambol para SyncML y DAViCal para CalDAV.

Si usar un servidor comercial no es un gran problema, la combinación de Google (para contactos y calendarios) y Toodledo (para tareas) puede constituir una buena alternativa. Al ser la opción que mejor funciona, nos centraremos en su desarrollo, intentando solucionar el problema de sincronización de tareas, no posible en este caso entre Evolution y el servidor.

Para ello, desarrollaremos una nueva aplicación de gestión de tareas, consistente en una aplicación web para iPhone, un plugin para Evolution y un servidor basado en CouchDB con una interfaz web para la gestión de dichas tareas.

Aplicación web

Para el desarrollo de la aplicación web se utiliza CouchApp, un conjunto de scripts que permiten construir aplicaciones basadas en CouchDB utilizando HTML y CSS para la parte de presentación, y JavaScript para la implementación de la funcionalidad. Esta aplicación utiliza la libería JQuery para realizar las consultas Ajax a la base de datos.

De esta forma se construye una interfaz web que permite añadir, eliminar y consultar tareas almacenadas en una base de datos CouchDB. Si la aplicación web es cargada desde el iPhone, se muestra una versión adaptada al dispositivo móvil, que hace más fácil la gestión de las tareas.

Plugin para Evolution

Para poder sincronizar las tareas entre Evolution y CouchDB será necesario el desarrollo de un nuevo plugin. Este plugin se basará en dos librerías ya existentes:

- Couchdb-glib: una librería basada en GLib que permite el acceso a bases de datos CouchDB.
- Evolution-couchdb: Evolution backend para el acceso a bases de datos CouchDB.

Será necesario programar el backend para calendario y, en particular, para las tareas, ya que la única parte implementada que existe es la de contactos.

Para añadir una nueva funcionalidad a Evolution primero hay que utilizar EPlugin. El objetivo es mostrar una nueva ventana de configuración cuando una lista de tareas sea creada, ofreciendo al usuario la posibilidad de utilizar CouchDB como sistema de almacenamiento. Este proceso se realiza en un documento XML y es detallado en el proyecto en la sección 5.2.2.2.

A continuación, hay que desarrollar el backend para calendarios. Para ello es necesario presentar primero EDS (Evolution Data Server). EDS gestiona el acceso a contactos, calendarios y tareas. Es un elemento CORBA que permite el acceso concurrente de varios clientes a la misma información con notificaciones de cambios. Puede ser extendido a través de la creación de plugins para diferentes tipos de fuentes de contactos/calendarios/tareas (en este caso en particular, CouchDB) mediante la programación de una librería compartida que será cargada al inicio de EDS.

Para la creación de esta librería para tareas almacenadas en CouchDB necesitamos desarrollar el backend para calendarios. Este backend se encarga de la comunicación entre EDS y la fuente de calendarios/tareas en particular.

En este caso tendrá que traducir las operaciones realizadas en Evolution (crear una nueva lista de tareas, añadir una nueva tarea, ...) a las operaciones necesarias en la base de datos y viceversa.

Replicación y gestión de conflictos

De esta forma podremos crear nuevas listas de tareas almacenadas en CouchDB. El último paso consiste en sincronizar la base de datos local del ordenador personal que ejecuta Evolution con la base de datos en el servidor. CouchDB ofrece la operación de replicación. Este proceso puedo ser automatizado y programado, en este proyecto se describe cómo realizarlo desde la interfaz web ofrecida por CouchDB.

Para finalizar, la aplicación tiene que implementar la política adecuada para la gestión de conflictos. CouchDB elige una versión como ganadora, dejando la versión perdedora almacenada hasta que se realice un proceso de compactado. Por lo tanto, la aplicación es la responsable de recuperar estos datos si la opción de fusión de los datos es posible, como se mencionó anteriormente.

Este proyecto fin de carrera desarrolla la parte de tareas, permitiendo la sincronización entre iPhone y Evolution de este tipo de información mediante el uso de CouchDB. Se propone como posible línea de trabajo a seguir el trasladar este sistema al resto de datos que componen la información personal (contactos, calendarios, notas, mails), de manera que todo trabaje en un sistema totalmente implementado en código abierto y con la posibilidad de gestionar nuestro propio servidor, sin depender de servidores comer-

ciales donde la privacidad y la gestión de nuestros datos no es la óptima.

List of Figures

Contents

Chapter 1

Introduction

In this chapter it is going to be described the motivations, objectives and the outline of the thesis.

In the motivation part, the problem of synchronization is presented. The increase of personal devices and the fact that connections are not always available make necessary the design of a synchronization system. Data must be consistent in every device.

In the objectives part, the aim of the thesis is presented. A synchronization solution between the iPhone and Evolution (a personal information manager for Linux) is looked for, trying to achieve it through open source protocols and with our own server. In the way to this solution, a missing link will be found, so a tasks lists application based on CouchDB is developed.

In the outline part the structure of the thesis is presented. Each chapter is briefly described and the outline is argued.

1.1 Thesis Motivations

Every user manages a series of information in his daily life. That is contacts, calendars, mails, tasks and notes. The problems arise when you have several devices, for example a mobile phone (iPhone) and a laptop, and each one of these devices manages these data with its own applications (Evolution on the laptop). It is evident that the user will like to have the PIM (Personal Information Management) data up-to-date in every device, that means to have them synchronized (see figure 1.1).

Synchronization is a fundamental component in every wireless network; data must be accordant when accessed by multiple users who are not always connected. Every data changed in one device should be reflected in all others as

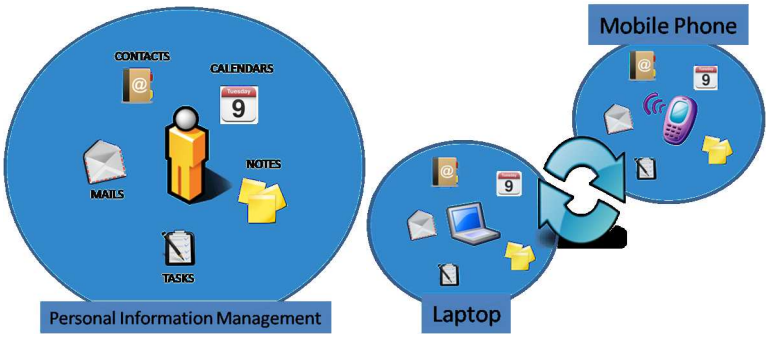

Figure 1.1: Personal Information Management Synchronization

soon as they are online. For example, when a phone number is added in someone's device, it should be reflected in the other devices contacts list.

Synchronization process is crucial nowadays because wireless devices are not always connected. Coverage is not universal, connections often get dropped and data contracts in mobile phones are still too expensive. These devices have to be able to work offline and synchronize with the world when they are back online.

There are several commercial solutions for the most common mobile phones, but they are sometimes quite expensive and they only work under certain conditions. I fix in this thesis the missing parts in the existing solutions for synchronization between iPhone and Evolution. The biggest challenges come from the iPhone with its closed operation system, making synchronization difficult based on open source software.

1.2 Thesis Objectives

The aim of this thesis is to design a system able to synchronize the PIM data between the iPhone and one of the most common desktop application in Linux, Evolution in GNOME. The intention is to use open source alternatives wherever it is possible.

To achieve this goal, the parts involved in the solution are studied. Those are the iPhone, Evolution and the tested servers. Then, an approach to the solution is made, describing which will be the ideal behaviour for a PIM synchronization system. Taking into account the limitations of the involved parts, a implementation of the solution is made, finding a missing link in this system. To fix that, a task lists application is built using CouchDB as storage system due to its good replication features.

The clients need to store the data locally in order to work offline. When they come back online, a replication process is needed between the local database in

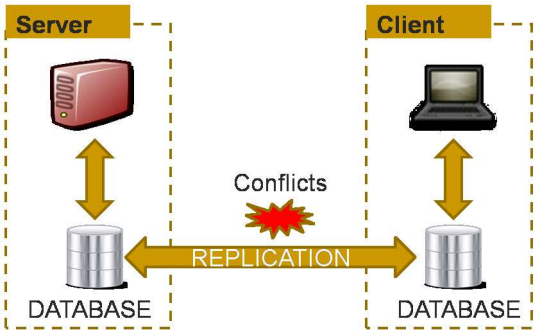

Figure 1.2: Replication as key of a synchronization system

the client and the database in the server. In this way it is achieved a consistent state in the data, sending each other the changes made meanwhile the connection was not possible. In this replication process, conflicts can occur. The same item has been changed in both instances of the database and a decission has to be taken in order to leave data in a consistent way (see figure 1.2). The optimal solutions in order to resolve these conflicts will be presented.

1.3 Thesis Outline

This section describes briefly the contents of each chapter:

Chapter 1: Introduction

The introduction chapter is divided into three parts. A general introduction is made and the motivations, goals and structure of the thesis are discussed.

Chapter 2: Literature Review

This chapter is composed by three parts: The "User requirements" part explains where the problem of synchronization comes from. The "PIM synchronization challenge" describes the typical problems found in the synchronization of contacts and other personal information. Finally, in "Standards" are presented the protocols involved in the implemented solutions.

Chapter 3: Solution Parts

This chapter is divided into four parts: "Mobile Phones Sync Capabilities" describes the synchronization protocols available for the iPhone and Android, "Desktop Applications Sync Capabilities" makes the same with Evolution, "Synchronization servers" presents the synchronization protocols supported by the tested servers and, finally, "Storage systems" introduces CouchDB comparing it with other kind of databases and arguing why it has been chosen for the tasks application to develop.

Chapter 4: Solution: Approach

The main challenge in a synchronization system is the conflict resolution. This chapter describes in its first part how this conflict resolution is optimal for a PIM synchronization system. The second part presents the limitations found in the involved parts in order to achieve this ideal system.

Chapter 5: Solution: Implementation

This chapter is divided into two main parts. The first one implements the synchronization systems with existing solutions, differentiating between commercial servers and our own server. In this implementation, a missing link is found regarding the task lists synchronization. In order to fix that, the second part explains the development of a task application based on CouchDB. This application is composed of a web user interface, accessing the database through a JavaScript, and a plugin for Evolution, accessing CouchDB with a task backend built for this purpose.

Chapter 6: Conclusions

The conclusions achieved in the development of this thesis.

Chapter 2

Literature Review

This chapter describes where the problem of contact synchronization comes from, what is already known about this problem and what other methods have been tried to solve it. Taking into account the user requirements and the synchronization challenges, the existing standards for synchronization will be presented focusing in the open source alternatives.

2.1 User requirements

The proliferation of mobile computing devices makes necessary the access to the data wherever the users are: in powerful and internet-connected machines usually in house or the office and where these powerful connections are not available, most frequently with a mobile phone or laptop outside, often only weakly or intermittently connected to a network.

Connecting to a network is not always possible. Very frequently, you can lose a call when entering inside a building or when you are driving by a highway. Data coverage is even worse. Moreover, mobile operators usually charge per byte or minute, so users will naturally prefer downloading once and syncing over paying per each access.

In this context appears the inherent problem of data synchronization. It is necessary to keep all the devices up-to-date and operational even when not online. Changes made when the device is offline should be reflected in the other devices as soon as the connectivity is available. You can observe a typical synchronization scenario in the figure 2.1. This sync session can occur in the background without the attention of the user.

To achieve this synchronization, a data synchronization protocol is needed. It must identify the changes quickly, resolve possible conflicts and propagate updates to various synchronizing devices. The information to synchronize include

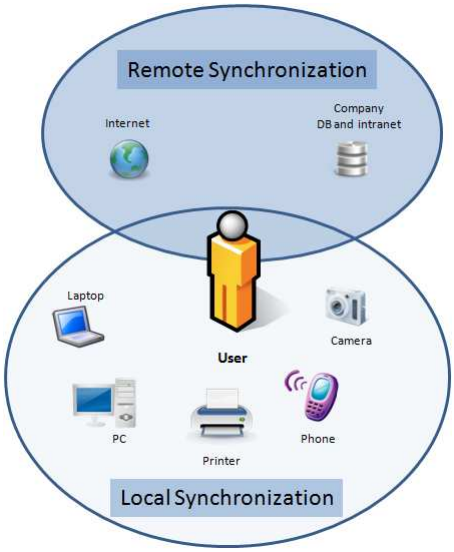

Figure 2.1: Typical sync required scenario

every kind of file available in these devices, as music, pictures or documents, but the most common requirement for users is to synchronize the personal information: that is contacts, calendars, todo/tasks, mails and notes. This kind of information presents several particularities that will be studied in the section 2.2.

2.2 The PIM synchronization challenge

To understand the challenge presented in PIM synchronization, a practical example is going to be studied: a list of contacts, one small piece of many applications and one of the main piece of our solution.

The first point to consider is how the data are stored: as a plain text file, a XML file, a database or in an object store. Being a list compound of several contacts, we are going to store it in separate records. Each record will be built with descriptive information about a contact: name, address, phone number, and so on. Here, the first problem comes up. There could be two contacts with the same name or with the same address. To differentiate them, a unique identifier is needed. So we aggregate a new field to each record with this identification. The information to store would be something similar to the table 2.1.

Database replication is a well understood problem, but PIM data is special in many ways. The challenge involves keeping the list of contacts correct as changes are made. Some changes are made on the server (if it exists) and some changes are made on a mobile device. Next, through some simple scenarios, it

Contact ID	Name	Address	Phone Number
102	Stefan Müller	Neuforgasse,13	0043611545454
215	Jane Smiths	Währinger Strasse,55	0043665464687

Table 2.1: Example of typical contact records

is going to be presented the most common problems in PIM synchronization.

ADDING A CONTACT

To achieve efficient data synchronization, only changes to the data must be sent - we want to send only the new records. For that, you need to pick out the new contacts from all the others. A possible solution would be to add a timestamp to the table and track the last time you synchronized.

In other way, the new record needs a Global Unique ID (GUID), and yet the mobile device does not have access to records being added by others users on other devices. To guarantee the uniqueness of the key value some solutions are possible. One mechanism is to use Universal Unique Identifiers (UUIDs) – long strings of alphanumeric values constructed from device-specific and time-specific data in such a way as to be guaranteed unique. Another way is to partition the set of possible keys across the mobile devices.

DELETING A CONTACT

This scenario presents even more difficulties than the last one. The mobile user deletes the record from the list of contact on her mobile device. That delete operation needs to be sent up to the other devices as soon as possible, but the record is no longer present to be sent.

To make delete action synchronization possible, you would need to hold a tracking table that keeps deleted rows around until the delete operation has been sent to the other devices. After a confirmation from these devices, the tracking table has to be cleared out in order to not send it again.

UPDATING A CONTACT

An update operation about the same contact has been made in two synchronizing devices, but the first one to update their local data is unable to synchronize his device until the next day, while the second one synchronizes immediately. When the first device synchronizes the next day, a conflict occurs. The correct behaviour would be not replacing the newer data (sent by the second one before) if we are talking about different devices of the same user. In other situation, a different rule could be applied.

The synchronization system needs to identify the fact that a conflict has occurred: the same record has been changed in two different mobile devices, and

just applying the changes in the order they are synchronized does not always do the right thing. It should take the right action to resolve the conflict: in this case, keep the later of the two times.

OTHER TASKS TO TAKE INTO ACCOUNT

This is a substantial list of tasks for a synchronization system to implement, but there are more. These scenarios were well-defined and with a set of rules. The devices share the same data format and all the conditions were ideal. In the real world, things change:

- Not always is possible a GUID. Many of the most common exchange formats for PIM data (as vCard 2.1/3.0 or vCalendar) don't include a mandatory UID property.
- Without a GUID, it is necessary to compare the content of items to identify matches. The problem arises when many different representations of the same information are allowed. Moreover, some devices are more limited than other ones. For example, a desktop software can support photo and several email addresses while a mobile is not able to support that.
- Previously to the synchronization, it is not possible to know which features are supported by the other peer, so any assumptions can be made. The solution must be interoperable with existing devices. The part more problematic could be when receiving an item back from an unknown peer. If a property is missing, has it been removed by the user or has the peer been unable to store it?

Moreover, in a real environment we will find many users, intermittent wireless networks, no access to the devices and so on, in comparison with a test scenario. Performance must be considered as well. The system must be optimized to scale up to many users synchronizing simultaneously, providing separate thread pools to manage database connections; configurable timeouts to ensure that resources are used properly and a long etcetera.

Finally, other features can be necessary too. Perhaps some changes are more urgent than others, and need to be synchronized with higher priority than the remainder; maybe secure authentication at each point in the chain is mandatory or data need to be encrypted.

2.3 Standards

This chapter presents the most important standards for data synchronization. These standards will constitute the basis for the solutions presented in the thesis. First, SyncML is introduced, being a common data synchronization protocol.

Then, WebDAV is described, an HTTP extension for editing and managing files collaboratively on remote web servers. For calendar, a special extension exists, CalDAV. It allows us to access scheduling information on a remote server. For storing this information, a special format is used, iCalendar, a standard to represent events, meeting requests and so on.

Data synchronization is defined as "the process of establishing consistency among data from a source to a target data storage and vice versa and the continuous harmonization of the data over time". To be successful in this process, the two sets of data need to be modified properly and a conflict resolution policy must be followed. A conflict occurs when the same data element is modified in both sets in an inconsistent way, being these conflicts the main challenge in the synchronization process.

For this communication process it is necessary a synchronization protocol. The main characteristics of this protocol have to be the following ones:

- Identify a particular database or a single record in a data base with the addressing of the data sets.
- Communicate modifications with standard commands defined for this goal.
- Required features, transport protocol and so on for exchanging these modifications.

In general, each company has used its own data synchronization protocols until now. These proprietary solutions only work in a subset of devices, being able to access only to a small set of networked data. The absence of a single synchronization standard poses many problems for end users, device manufacturers, application developers and service providers. With this purpose - the development and promotion of a single, common data synchronization protocol used industry-wide – SyncML was born.

2.3.1 The SyncML Protocol

SyncML is a specification for a common data synchronization framework based on the exchange of XML-based messages between networked devices. SyncML is sponsored by companies like Motorola, Nokia, Ericsson, IBM and many others.

The SyncML specifications include:

- A representation protocol, which defines the XML messages format.
- A synchronization protocol which defines how SyncML messages shall be combined together in order to accomplish a synchronization session.
- The binding for different transport protocols (HTTP, OBEX).

- A protocol for device management

The typical scenario is a client-server architecture, where the client contains a sync agent and usually sends its modifications first. The server implements both the server-side synchronization agent and the synchronization logic including procedures to interpret the modifications, to discover and manage conflicts and to generate return messages.

SyncML defines seven synchronization modes:

- Two-way sync: the most common mode whereby client and server exchange modifications on their data.
- Slow sync: mode used for recovering from an inconsistent state. All the client database records are compared, field by field, with the ones on the server.
- One-way sync from the client only, from the server only.
- Refresh sync from the client only, from the server only: in these modes the client or the server sends its entire database which replaces its database in the other.
- Server alerted sync: the server sends an alert to the client notifying it of the mode to be used.

Other basic concepts about SyncML are the following ones:

Synchronization anchors: used to know which was the last synchronization successfully completed. Two anchors are needed: the last anchor representing the last sync performed and the next anchor representing the current sync. These anchors are exchanged at the beginning of the session. The server can detect if the last synchronization encountered problems comparing the given Last with the stored anchor. If they do not match, client and server are out of sync.

ID Mapping: server and client usually use a different ID for the same item, so the server needs to keep a mapping table between the client Local Unique IDs (LUID) and the server Global Unique IDs (GUID).

Conflicts: the same item has been modified on both server and client. The client is notified of the error condition by means of a status code.

Security: SyncML allows two authentication methods: basic and MD5.

Addressing: entities involved in synchronization (databases, items, etc) can be addressed through a named convention based on URIs (Uniform resource Identification)

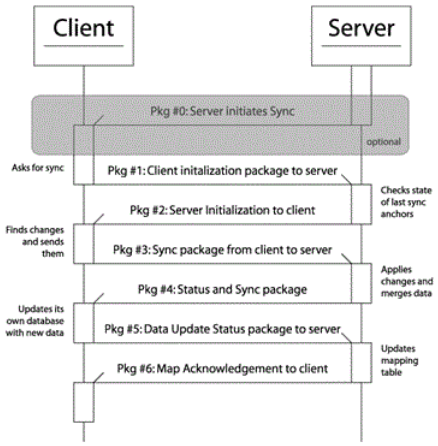

Figure 2.2: SyncML Synchronization Process

Device capabilities: the server can apply proper optimizations according to the remote device resources and features (e.g. database types, available memory, and transmission speed). They should be sent only if requested.

The complete SyncML synchronization process requires three steps (see figure 2.2[1]):

- The initialization where device characteristics and synchronization requirements (as type of sync, protocol features supported, which database needs to be synced) are sent. Packages from #0 to #2 in the figure.
- The modifications sent as a well formed XML document, with a header (including the routing info, database addressing and protocol versioning) and a body (with one or more sync commands). Packages from #3 to #4.
- The client mapping step, composed of the packages #5 and #6, updates the mapping table in the server.

2.3.2 WebDAV

WebDAV means Web-based Distributed Authoring and Versioning. It is a set of extensions to HTTP (Hypertext Transfer Protocol) to allow users to edit and manage files collaboratively on remote web servers. It is a standard defined in RFC 4918 with an incremental update defined in RFC 2518.

Its main features are:

[1]Source: http://max.berger.name/research/syncml/syncmlTimeline.png

- Locking: for overwrite prevention.
- Properties: information about the author, modified date, etcetera.
- Name space management.
- Collections: creation, removal and listing of resources.

The protocol defines new methods and headers for HTTP. For example, `COPY` for copying one resource from one URI to another or `LOCK` to put a lock on the resource. You can find all the added methods and headers and its function in the RFC mentioned above.

This standard has many implementations in Linux. For example, KDE has native WebDAV support. This means that Konqueror or any other KDE application can access to a WebDAV server without external components.

Among the different existing extensions, the most useful ones for this thesis are:

- For calendaring, CalDAV: A protocol allowing calendar access via WebDAV. CalDAV models calendar events as HTTP resources in iCalendar format, and it models calendar containing events as WebDAV collections.
- For groupware, GroupDAV. It allows client/server groupware systems to store and fetch objects (calendar entries, contacts ...) instead of web pages.

Next, CalDAV and the iCalendar format will be studied in more detail because they are main pieces in the solutions shown in this thesis.

2.3.3 CalDAV

As I mentioned above, CalDAV is a calendaring extension to WebDAV. It is an Internet standard, defined in RFC 4791, allowing a client to access scheduling information on a remote server. Multiple clients can access to the same information, so cooperative planning and information sharing is possible through this standard.

The format used for storing the data is iCalendar. You can find more detailed information about this format in the section 2.3.4.

The architecture followed by CalDAV consist on data (events, tasks, free-busy, info and notes) stored in directories (collections) where multiple items (resources) reside.

Among the existing implementations, we can find it working on the iPhone (firmware 3.0 and forward versions) and Evolution.

2.3.4 iCalendar

iCalendar is a standard, defined in RFC 5545, for representing meeting requests, tasks or any other kind of scheduling information. Its main purpose is the sharing of calendars, being independent of the transport protocol (email, WebDAV, SyncML).The extension used for this file format is `.ics`.

The first line of an iCalendar object must be `BEGIN:VCALENDAR`, and the last line `END:VCALENDAR`. The body is made up of a list of calendar properties and one or more calendar components. An example for a task could be:

```
BEGIN:VCALENDAR
VERSION:2.0
PRODID:-//ABC Corporation//NONSGML My Product//EN
BEGIN:VTODO
DTSTAMP:20100130T134500Z
SEQUENCE:2
UID:uid4@host1.com
ORGANIZER:MAILTO:unclesam@us.gov
ATTENDEE;PARTSTAT=ACCEPTED:MAILTO:jqpublic@example.com
DUE:20100415T235959
STATUS:NEEDS-ACTION
SUMMARY:Submit Income Taxes
END:VTODO
END:VCALENDAR
```

It is a task due on April 15, 2010 with summary *Submit Income Taxes*, and *jqpublic@example.com* as attendee. The field *SEQUENCE:2* means the element has been modified twice since it was created.

Chapter 3

Solution Parts

The development of a solution for this thesis involves three main elements. In this chapter, they will be presented and studied, detailing their principal features.

First, we have a mobile phone. The solutions will be focused on iPhone but Android will be considered as well. Second, we have a desktop application to manage our personal information. Here, Evolution will be the chosen one. To achieve the PIM synchronization between these two elements, the mobile phone and the desktop application, a third system is required. I will focus on a solution based on our own server, without depending of third-party companies, but commercial solutions will be presented as an alternative of them when they do not complete all our requirements.

For this synchronization system based on our own server, three alternatives exist in order to store the information. There will be compared the relational databases, the object-oriented databases and the version management systems. CouchDB, a object-oriented database, will be the chosen one for the task application because of its good replication features.

Figure 3.1: Solutions parts: mobile phone, desktop application and the sync system.

For the communication between devices, the standard protocols presented in the literature review, SyncML and CalDAV, will be used.

So let's start with the synchronization capabilities of our two main elements, the mobile phone and the desktop application.

3.1 Mobile Phones Sync Capabilities

The iPhone does not have a full and intuitive solution for PIM synchronization. The different personal information (contacts, calendars, tasks ...) have to be synchronized by means of different methods. This section presents the capabilities this phone offers us, completing it with the possibilities for Android, where the openness in the operating system results in an easier synchronization process.

3.1.1 iPhone

The possibilities to synchronize contacts and calendars in this device are very limited, even more if the intention is to make it using open source solutions. To describe the capabilities for synchronization in iPhone, I am going to distinguish between native capabilities and capabilities achieved by third-party applications. Take into account that some of these features are only available for iPhone OS 3.0 and forward versions.

NATIVE SOLUTIONS

If you go to `Settings→Mail, contacts, calendar →Add account` in your iPhone, you will find the following options related with synchronization (see figure 3.2):

- For contacts and calendars:
  - **Microsoft Exchange:** It uses the Exchange ActiveSync protocol which provides push synchronization of contacts, calendars and emails between ActiveSync-enabled devices. It is a proprietary protocol, but licensed to some mobiles companies as Apple for iPhone or Google for Android. This feature can be used to synchronize the iPhone with any Microsoft Exchange server or with your Google account. There is an open source alternative, OpenXchange, but the plugin needed in the server for making it work with the iPhone is not available for the community version, the paid version is required.
  - **Mobileme:** a paid subscription-based collection of online services and software offered by Apple. It includes storage, address book, calendar, gallery, web site hosting and some other services.

Figure 3.2: iPhone Settings for Contacs, Mail, Calendars, ...

- For contacts in `Other`:
  - **LDAP** (Lightweight Directory Access Protocol): an application protocol for querying and modifying data using directory services running over TCP/IP. It is a service very used in companies to have access to the enterprise address book, but it is not suitable for synchronization because it is not able to work offline.
- For calendars in `Other`:
  - **CalDAV**: this standard protocol has been presented already in the section 2.3.3 on page 27. It allows users to synchronize their iPhone calendars with a CalDAV server.
  - **Subscribed calendar**: it adds events from a public calendar. Only one-way synchronization, not valid for a full synchronization solution.
- For mails: there are several options. You can configure accounts for Gmail, Yahoo mail, AOL and many others.
- For notes: the native application is very simple. It does not allow any kind of synchronization natively. Through iTunes it is possible to sync them with MAC OS X Mail or Microsoft Outlook.

To sum up, the native capabilities are much reduced. The only possibility related with open source appears in calendars, with the use of the standard protocol CalDAV. If we relax this condition, the open source alternative, the free option could be the use of Google to synchronize contacts and calendar through Microsoft Exchange. If you need other alternatives, you can use OpenXchange together with its plugin for iPhone or Mobileme from Apple, paying the required

amount. For mails the solution is almost complete. However, there are not native support for todo lists on this phone.

THIRD-PARTY APPLICATIONS

To complete the iPhone synchronization capabilities, some applications have been built. However, due to Apple's restrictions policy, these applications are very limited.

- For contacts:
  - **SyncML**: this standard protocol for synchronization has already been described in the subchapter 2.3.1 on page 24 of this thesis. There are several applications that let you to synchronize your contacts in the iPhone with a SyncML server. For example, Funambol or Synthesis.
- For calendars:
  - **SyncML**: until the iPhone SDK 4, no calendar API was available. At the moment of writing this thesis, the iPhone OS 4 is still not in the market, so the existing applications are not able to synchronize with calendar in iPhone.
- For tasks/todo:
  - There are dozens of todo applications available for iPhone. Some of them, as Toodledo, let you to synchronize tasks with Microsoft Outlook, Apple's iCal and other online todo lists. However, for Evolution the synchronization can be made only in one way, from the cloud to Evolution. This missing link will be fixed with the creation of a task application able to synchronize with Evolution in the two ways. For more details, go to the section 5.2 on page 50.

To conclude, at the moment only contacts can be synchronized with third-party applications in the two ways. With the task application developed in this thesis and the openness of the calendar API (on the iPhone OS 4), the possibilities for a complete synchronization in the iPhone are going to be increased.

As it has been detailed, there is not a simple way to synchronize your personal information in iPhone. The only open source alternative appears with the use of a SyncML client for contacts and the configuration of CalDAV for calendars. Let's compare it with other of the most famous operating systems for mobile phones at the moment, Android from Google.

	WebDAV	LDAP	CalDAV	Google	Others
Mail				x	IMAP, POP, Exchange, ...
Contacts	x	x		x	
Calendar	x		x	x	
Tasks	x		x		
Memos	x		x		

Table 3.1: Synchronization capabilities for the different resources in Evolution.

3.1.2 Android

Android is an open source operating system and software stack for mobile devices, being the project led by Google. The fact to be an open source solution involves not to find so many restrictions as in the iPhone OS, although the system is quite oriented to use Google services.

Contacts, calendars and mails are synchronized automatically with your Google account. This synchronization with Google services can be completed with an application for the synchronization of tasks (e.g. gTask).

If you prefer to store your data in an own server instead of Google servers, the use of a SyncML client is an option. For example, Funambol is an open source and quite complete server. It lets you to synchronize contacts, calendars, tasks and notes with any SyncML server.

To finallize with the most common standard protocols for PIM synchronization, the support for CalDAV at the present is not so good in Android. There is an application, Calendar (CalDAV) Sync for Android by Hypermatix, with some problems at the present[1].

3.2 Desktop Applications Sync Capabilities

The synchronization capabilities of two of the most famous operating systems for mobile phones at the present, iPhone OS and Android OS, have been presented in the section 3.1 . Now, I am going to present the synchronization solutions that desktop applications in Linux offer us to find common points between them.

Let's start with the official personal information manager and workgroup information management tool for GNOME, Evolution. It is composed for mail, calendars, contacts, tasks and memos. The table 3.1 collects the synchronization capabilities for each one of these resources.

Those are the possibilities to synchronize Evolution with other devices in the version 2.28.3 of the application. The synchronization with Exchange servers

[1] More details in `http://wiki.davical.org/w/CalDAV_Clients/Calendar_Sync_for_Android`

can be made with the use of WebDAV as well. Moreover, with the installation of specific plugins other alternatives can be used. The plugin system for Evolution will be discussed in detail in the section 5.2.2.2 on page 56, where an extension for this application will be built to allow synchronization with the task application to be created in this thesis.

There are also external tools, like SyncEvolution, to synchronize data with Evolution. It is able to synchronize contacts, appointments, tasks and memos with a SyncML server using this standard.

3.3 Synchronization Servers

Now that we know the synchronization capabilities of iPhone and Android in the mobile phone side, and Evolution in the desktop application side, we have to find the common protocols that let us to achieve the synchronization between these devices. We differentiate here between solutions based on our own server where the information resides on it and commercial solutions where our information is in an external server, no administrated for us.

Finally, with the available alternatives we will be able to propose a solution and test in different scenarios in the following chapters.

3.3.1 Based on own servers

Due to the limited options presented by the iPhone, the only chance to synchronize contacts with Evolution using our own server is through a SyncML server. There are several open source servers supporting SyncML at present. In this thesis eGroupware is going to be tested , a groupware server with support for SyncML and CalDAV.

For calendars, the only chance to access iPhone appointments is through the CalDAV protocol, so we need a server with CalDAV support. There are specific servers, like DAViCal, or other kind of servers which are able to work with this standard, like eGroupware mentioned above.

Once the address book and the calendar information is in the server, the synchronization options with Evolution are:

- **GroupDAV:** it uses WebDAV to access to the contacts and calendars in the server. It is not useful because it only works when a connection is available, not offline.
- **SyncEvolution:** an external tool to synchronize PIM via the SyncML protocol.

Relaxing the open source requirement, an OpenXchange server could be a good option. You need to acquire the paid version and install the extension called "OXtender for Business and Mobility". The synchronization will be made through the ActiveSync protocol, a no open source synchronization protocol, as it was mentioned in the iPhone capabilities chapter. However, this solution let you have your own server and synchronize contacts and calendar in an easy way. You can find more information and prices on the next link:

http://www.open-xchange.com/en/mobility-solutions-en

3.3.2 Commercial servers

There are many options to store your PIM data in commercial servers, most of them are not free. For the scenario required, with the iPhone and Evolution as devices to synchronize, a good option is Google services.

Contacts, calendars and mails in your iPhone can be synchronized with your Google account using the ActiveSync protocol, without requiring a third-party application running in your device. The configuration process is described in the section 5.1.2.1 on page 48.

Google also offers a task service integrated with its calendar. There is a customized web page when access the service with the iPhone. However, if you want to work offline and synchronize when the connection is available, you need an external application (e.g. GeeTasks).

The problem with this solution is the synchronization between Google servers and Evolution. This is based on a plugin for Evolution that only works online, not being possible to manage your calendars and contacts when no connection is available.

3.4 Storage Systems

Up to this point of the thesis, only protocols for the communication of the changes between devices have been presented. Other important point to take into account is the storage system used in each one of these devices. In this section, CouchDB, the object-oriented storage system chosen for the task application to be built, will be described comparing it with relational databases and version management systems.

This comparation is focused on the synchronization features these systems presents. CouchDB has been chosen because of its good replication process. For the goal of this thesis, the synchronization of PIM, replication is more important than data structure. So CouchDB will be detailed arguing its advantages in comparison with other systems.

CouchDB has been designed to build modern software applications that tend to be web-based, document-oriented and distributed in nature.

During the last decades, relational database management systems have been used to build almost every application. There are lightweight, inexpensive and well-document solutions with this model - like the open source MySQL - used in millions of web sites. Its main characteristic is a strict schema-based structure. This is ideal for some scenarios – like transaction systems – but it is not optimal for new projects like blogs, wikis or forums which fit better with a document-oriented database.

The main features of CouchDB are:

- Document-oriented database management system.
- Javascript-based view model for aggregating and reporting on data.
- Incremental replication with bi-directional conflict detection and resolution.
- Developed in Erlang OTP, a programming language with excellent concurrency features.

These features are going to be described in more detail as follows:

DOCUMENT-ORIENTED DATABASE

The database is composed of documents. Each document is made up of fields and values. These values can be strings, numbers, dates, boolean or even lists of objects. Attachments can be included as well. For example, a document representing a task could be similar to this one:

```
"Summary": "Buy milk"
"Due date": "1.June"
"Description": "Go to the supermarket and buy 2 litres of milk"
```

There is not a strict schema, each document can be different. The relational model is based on a set of tables, each of them with a predefined schema. In CouchDB that does not exist. If another task does not have "Due date" this field can be omitted and a "NULL" value is avoided. In addition, if any change is necessary in the structure of a relational database, a lot of difficulties appear due to dependencies and integrity issues. All of this is not a problem in a document-oriented database.

Needless to say, this model presents some disadvantages too, like the lack of a defined structure and unnecessary replication of data across documents. However, the performance of relational databases can be almost modeled with CouchDB. For example, entity relationships[2].

[2]Visit this link for more detail:
http://wiki.apache.org/couchdb/EntityRelationship

Each document has a unique ID and metadata maintained by the database system, like the revision number. The documents are stored in the database on a flat address space.

JAVASCRIPT VIEW ENGINE

To recover unstructured data and show them in a useful and interesting way, views were designed. Views are the method of aggregating and reporting on documents in a database. They are defined inside a design document and are replicated too. For an optimal performance, requested views are up-to-date incrementally.

REPLICATION AND CONFLICT DETECTION AND RESOLUTION

CouchDB is a peer-based distributed database system. Documents and designs can be replicated, so full database applications can be distributed and maintained up-to-date when connectivity is possible.

CouchDB peers do not have to be configured or tracked. No record is kept of which peer any particular document or revision come from. That is an important difference with version management systems, like git, where each peer has to be explicity tracked.

This replication is incremental again to improve the performance of the system. The process only examines documents updated since last replication. For each document updated, only changed fields are sent across the network. Partial replicas are possible too, filtering with a JavaScript function.

In case of conflict in the replication process, each database instance deterministically decides which document is the winner. Only this document can be shown in views, while losing conflicts remain in the database until a deleted or purged operation during the compaction.

To resolve the conflicts several options are possible: manually, depending on the nature of the data and the conflict or by automated agents. If resolved conflicts result in more conflicts, the system accommodates them in the same manner, determining the same winner and maintaining document semantics.

OTHER INTERESTING FEATURES

CouchDB has not locking mechanism. The first to commit succeed. The next ones get a conflict error, the latest revision is shown and they have the possibility to make the changes again.

The documents are not overwritten. There are a main revision and the others are stored for archival purposes. That is similar to a revision control system. However, the programmers cannot assume the existence of this history because the compact option used to recover some wasted space deletes them.

As it could be expected, CouchDB implements the ACID (Atomicity, Consistency, Isolation and Durability) properties.

To complete all this features, a RESTful HTTP API is available. It lets any platform that supports HTTP requests to interact with CouchDB without any client library to be installed.

To make the administration easier, a web-based administration tool is available, Futon. From there it is possible to create, modify or delete databases and documents. Compaction, replication and test are possible using an easy web user interface. You can access it in your local installation with the next URL: `http://127.0.0.1:5984/_utils/`.

Chapter 4

Solution: Approach

This chapter describes an approach to the ideal solution for the required scenario. This ideal solution presents the following main features:

- Synchronization of every PIM data: mails, contacts, calendars, tasks, memos.
- Based on open source software.
- The information resides in an own server.
- Automatically synchronization of the data when coming back online.
- Simple configuration process.

These are desired features for the system to implement. Together with these, there are other many features like security, fail tolerance, scalability and a long etcetera. But the main challenge in the synchronization process and the basis of a synchronization system is the conflict detection and resolution. When a conflict is detected, several options are possible. This chapter compares these posibilities depending on the context. To conclude, the known limitations of the involved parts are described in order to implement a real solution.

4.1 Conflict Resolution

This challenge of conflict resolution in a synchronization process is going to be described in this section with a simple example. The most suitable solutions when a conflict is detected in PIM synchronization are mainly two: merging and the "newer wins". Find more details below.

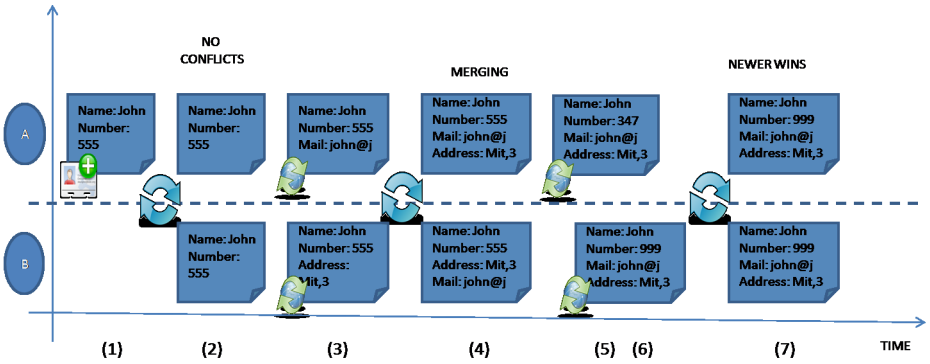

Figure 4.1: Conflict resolution time line

The scenario is composed of two devices: A and B which want to have their PIM data synchronized. The device A creates a new contact (V1) with the fields `Name:John` and `Number:555` in the **time slot (1)**. When it comes back online, it synchronizes with the device B, so now both devices have the new contact in **time slot (2)**.

4.1.1 Merging

During the **time slot (3)**, the device A updates the contact adding a new mail field (V2a). The device B updates it too with a new address field (V2b). When they are able to synchronize again, a conflict has to be detected. The same element has been changed in two different devices. For this particular situation the best solution is to merge the elements, because the changes made are not incompatible, as it is shown in the **time slot (4)**. If we had chosen only one of the modified versions as winner, we would have lost information added by the user. That had not been the optimal solution.

4.1.2 Newer wins

In the case above the changes were compatible, but what happen if the changes cannot be merged? For example, the device A changes its number to `347` in the **time slot (5)**, while the device B changes its number to `999` in the **time slot (6)**. When they sinchronize again, a conflict occurs. But this time merging is not possible because the field changed is the same in both devices. For the resolution of this conflict, we need a time stamp in order to know which one was the last in applying the changes. The solution that fits best would be to resolve the conflict with the newer as winner. The last change made by the user in its devices would be the desired to remain after the synchronization.

To sum up, the conflict resolution for PIM data will consist on merging if it is possible or the "newer wins" policy if merging is not possible.

4.2 Limitations

It has been described above the desired behaviour for a synchronization system, but the implementations already available present several differences with regard to this ideal situation. The main limitations in the clients and the servers studied in this thesis are detailed next.

4.2.1 Limitations due to the clients

4.2.1.1 iPhone

This mobile phone does not offer native applications for all the PIM required for a user. There are mails, contacts, calendars and memos applications, but there is not a task list application, so a third-party one is needed.

Regarding the native synchronization capabilities, there are no possibilities for tasks and memos. Moreover, if you want to use SyncML as synchronization standard, you have to run a third-party application, so the process is not automatic and it has to be started by the user. Other limitation regarding SyncML comes from Apple's policy avoiding third-party applications to access to the calendar (until the iPhone OS 4). This restriction only leaves CalDAV as possibility with open source alternatives.

To finish, the iPhone does not add a `UID` and `Time Stamp` fields to the contacts created. The `UID` is needed for the identification of a particular item and the `time stamp` for the conflict resolution, to know which one was the newer change. Observing the logs generated by the SyncML clients in the iPhone, you can check that any of these fields are presents. The vCard file generated by the application only contents the most common fields for this kind of PIM, but any information for a good conflict resolution. This limitation makes the conflict resolution in the server in an optimal way almost impossible.

4.2.1.2 Evolution

When using eGroupware as server, a option to load the PIM data into Evolution is with the GroupDAV connector. However, this option only works when a connection is available, showing an error if an update is made offline. Therefore this alternative has to be rejected, because it does not suit with synchronization requirements.

The only option to synchronize with open source software is SyncEvolution. This application works pretty well with SyncML, but it is not integrated in

Evolution and the user has to run it every time he wants to synchronize his data.

If you choose Google services, the plugin available for Evolution only works online, being not possible to manage your calendars and contacts when being offline.

4.2.2 Limitations due to the servers

The tested servers don't follow the rule mentioned above for the conflict resolution, where the best option is merging when it is possible and the "newer wins" policy when merging is not an option.

The eGroupware server can be configured to manage the conflict with SyncML between the following alternatives:

- The client wins conflicts and overwrites the server data.
- The server wins conflicts and overwrites the client data (by default).
- Merge data
  - the client can only add new information to Contacts
  - the client can only change attendee's state or append new information to Events
  - deleted entries will be added by the server again
- Duplicate entries of conflicting entries are created from both versions.
- The client can never change server data but have its own version.
- The sever reverts all changes the clients report.

As you can observe, the merging option is very limited. For example, if you delete a contact from the iPhone and synchronizes it, the contact appears again in your phone, because deleted entries are added again by the server for the merging conflict resolution option. Moreover, no "newer wins" policy exists, so even in the case clients add a time stamp, not conflict resolution in this way is possible.

Other of the tested servers, the SyncML Funambol server, doesn't support this policy as well. The only options for conflicts resolution are where the client/server wins and the merging policy.

The best alternative regarding the conflict resolution seems to be Google. Merging is well supported and where merging is not an option, Google duplicates the conflicting field keeping both values. Evolution is able to show a field with several values, but the iPhone only shows one of the values of each field.

To sum up, the existing options don't manage the conflict resolution in an optimal way for PIM data. On the one hand, the clients don't add the required fields (time stamp and UID) to the created items. On the other hand, tested servers don't support the "newer wins" policy and merging presents several problems with unnecessary replication and loss of data as result.

Chapter 5

Solution: Implementation

The solution proposed in this thesis is based on the union of different existing solutions. The goal is to achieve a synchronization system able to synchronize the PIM data between the iPhone and Evolution. Two possibilities are taken into account: one based on our own server and other one where the information resides in an external server.

A missing link is found in this system. There are not available solutions for a two-way synchronization of tasks between iPhone and Evolution. A task application is built to fix this missing link. It is based on CouchDB, with a web application accesing to the database via JavaScript and a plugin for Evolution. The plugin adds to this application the capacity to store task lists on a CouchDB database.

5.1 Synchronization Systems

This section describes the implementation of the synchronization systems working with Evolution and the iPhone. As I mentioned above, two alternatives are considered. If you need to keep your data out of external servers and you want to have the possibility to manage your PIM data when you are offline, the section 5.1.1 describes how to implement it. However, if that is not a big problem, the section 5.1.2 describes a more simple system to implement based on Google services.

5.1.1 Our own server: SyncML and CalDAV

Several combinations of the available tools work together for achieving a basic synchronization between the devices, although all of them are based on the same standards: SyncML for contacts and CalDAV for calendars (tasks and

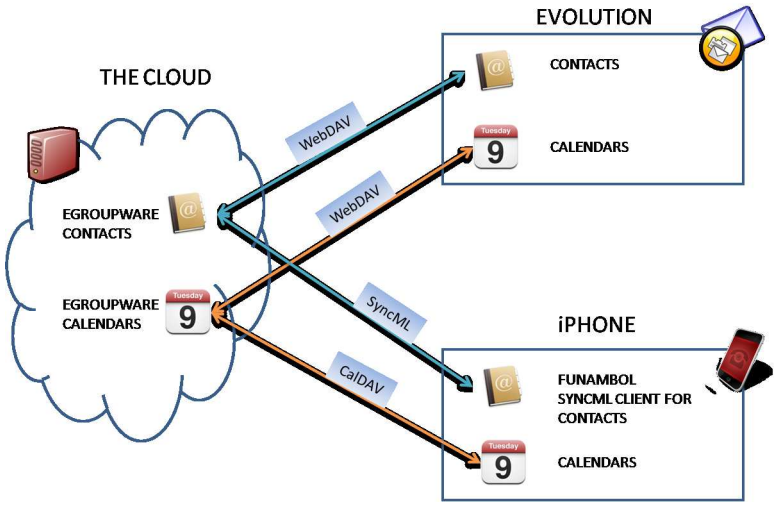

Figure 5.1: Synchronization system based on our own server

memos are not supported natively for iPhone). Next it will be described the process to configure the three elements of the system: the server, the iPhone and Evolution.

5.1.1.1 Server: eGroupware or Funambol+DAViCal

The easiest option is to use a Groupware server. This kind of servers usually support SyncML and CalDAV, with a complete web user interface for the management of PIM. In this thesis I will use eGroupware because it is very easy to install and use.

Other alternative is to use a SyncML server (e.g. Funambol) and a CalDAV server (e.g. DAViCal) connected between them (e.g. CalDAV connector for Funambol). However, this option requires more configuration and it is more complex that the option above.

The eGroupware version used in this thesis is 1.6.003. To install the package for a Ubuntu/Debian distribution, add the following content to your `/etc/apt/sources.list`:

```
deb http://download.opensuse.org/repositories/server:/eGroupWare/[Distribution]/ ./
```

Replace **[Distribution]** with Debian_5.0, xUbuntu_8.04, xUbuntu_9.04 or xUbuntu_9.10. Update the repositories and install it:

```
$ sudo apt-get update
$ sudo apt-get install egroupware
```

For other distributions there are packages as well. The installation process is described in the next link: `http://www.egroupware.org/wiki?wikipage=ManualSetup`. Basically you should go to the URL: `http://localhost/egroupware` and follow the instructions on the web.

eGroupware installs Postgres as default database. However, other relational databases can be used as well. I will use MySQL because it has an easier configuration process. You may install the necessary packages with:

```
$ sudo apt-get install mysql-server mysql-client php5-mysql
```

Restart apache2 to load the new extension for PHP and you will be now able to choose MySQL as database in your eGroupware installation.

If everything has gone alright, you can login in the web user interface and manage your contacts, calendars, mails, etc. The user should have the "syncml" and "groupdav" options activated by the administrator in order to synchronize the PIM.

In the next section, the iPhone will be configured to synchronize contacts and calendars with this server.

5.1.1.2 iPhone: Funambol SyncML client and native CalDAV

As it was mentioned in the section 3.1.1 on page 30, the only possibility to synchronize contacts with a open source alternative in the iPhone is the use of a SyncML client. There are several free alternatives in the App Store as Funambol or Synthesis SyncML. I will use Funambol because of its facility to be configured.

For calendars, the CalDAV native support will be used to synchronize with our eGroupware server.

CONTACTS

Open the Funambol application and go to `Settings`. In `Account` you can find the following fields:

- **Server:** http://[your IP or domain name]/egroupware/rpc.php
- **Username:** the name of the user you want to synchronize its PIM.
- **Password:** the password for the user typed above.

Coming back to `Settings`, you find `Contacts`. Enter and configure the following fields:

- **Sync Direction:** you can choose between "Two-Way", "Server to phone" and "Phone to server".
- **Remote Name:** ./contacts

Everything is ready now for the first synchronization. Go to the first screen and push "Sync Contacts". A "Synchronization successful" message appears is all has gone alright. If there are some errors, you can find more information in `Settings→Log`.

CALENDAR

As it was shown in the figure 3.2 on page 31, the iPhone has native support for a CalDAV server. Go to `Settings→Mail,Contacts,Calendar→Add Account...` and choose `Other→Add CalDAV Account`. A configuration window appears with the following fields:

- **Server:** the IP or domain name of the eGroupware server.
- **User Name:** the name of the user you want to synchronize its PIM.
- **Password:** the password for the user typed above.
- **Description:** a optional description of the server.

Push `Next` and the iPhone will try to verify the CalDAV account information. If a message appears asking if you want to try setting up the account without SSL, push `Continue`. The verification fails because "Advanced Settings" are required. Enter there and type the next configuration parameters:

- **Use SSL:** OFF [1]
- **Port:** 80
- **Account URL:** `http://[server IP]/egroupware/groupdav.php/[user name]/calendar`

There is an application for the iPhone (not free, not open source) to manage the information stored in a eGroupware server. Its name is iGroupMind. An extension in the server is required in order to make it work.

5.1.1.3 Evolution: GroupDAV and SyncEvolution

This section explains the necessary steps to synchronize your eGroupware server with Evolution. As it was mentioned in the Evolution Sync Capabilities section, we will use SyncEvolution.

[1] You can configure the server to use SSL if it is required.

SyncEvolution is a command-line application that works with SyncML as protocol to achieve the synchronization between the eGroupware server and Evolution. It is able to synchronize contacts, calendar, tasks and memos. A graphical user interface called "Sync" is available. Install this application with the following command:

```
$ sudo apt-get install syncevolution sync-ui
```

The easiest way to configure it is with the graphical user interface. Open the application and go to the section `Change or edit sync service`. There add a new service with the following options:

- **Name:** eGroupware
- **User Name:** the name of the user you want to synchronize its PIM.
- **Password:** the password for the user typed above.
- **Server address:** http://[server IP]/egroupware/rpc.php
- **Contacts URI:** ./contacts
- **Appointments URI:** ./calendars

Once eGroupware has been configured, you can push the `Sync now` button and the synchronization will take place.

5.1.2 Commercial servers: Google and Toodledo

If the option to store your PIM in an external server is not a big problem and when you use Evolution you have a connection available, there is another alternative for the synchronization of the iPhone and Evolution. It is based on Google for contacts and calendars, and Toodledo for tasks (see figure 5.2).

You need an account in Google and Toodledo for this process. The configuration in the iPhone and Evolution is described in the next sections.

5.1.2.1 iPhone

For contacts and calendars synchronized with Google, go again to `Settings→Mail, Contacts,Calendar→Add Account...`, but this time choose `Microsoft Exchange` as type of new account. This will use the ActiveSync protocol for the synchronization. Enter the following parameters:

- **Email**: your full google email account address.

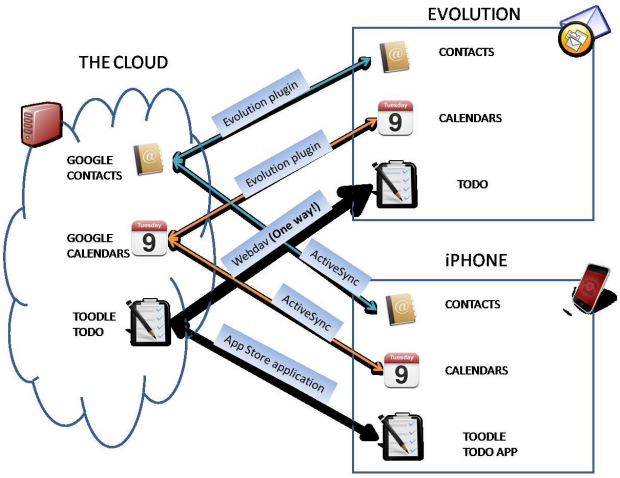

Figure 5.2: Solution based on commercial servers

- **Username**: your full google email account address.
- **Password**: your google account password.

Tap `Next`. If a "Unable to Verify Certificate" message appears, choose `Accept`. When a new Server field appear,enter `m.google.com` and tap `Next` again. You will be asked about which Google services you want to synchronize with your phone (Mail, contacts and Calendars).

For tasks synchronization with Toodledo, an application (not free) with this name exits for this phone in the App Store. Remember that the iPhone does not support tasks lists natively, so an external application is needed.

5.1.2.2 Evolution

This PIM desktop application supports Google services thanks to a plugin already included in the version used in this thesis. Just add a new address book and calendar with your google account user name and password in Evolution, but take into account that you can only manage your personal information when being online.

For synchronizing Toodledo and Evolution, first you have to enable the live iCal link in `Tools&Services→iCal` at the web page. They give you a URL you should enter in a new WebDAV tasks list.

If you have a look again to the figure 5.2, you can observe the synchronization between the Toodledo server and Evolution is only one-way, from the server to Evolution.

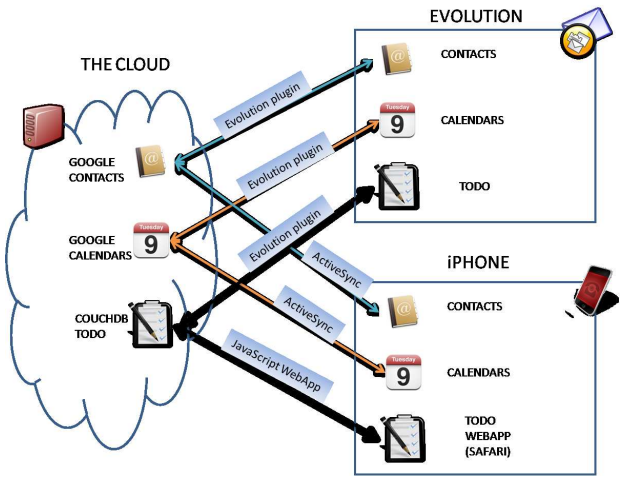

Figure 5.3: Solution based on CouchDB for tasks

To fix this missing link, a new task manager is going to be created based on CouchDB as storage system (see figure 5.3). This program will be the basis for a future improvement in the graphic above, making contacts and calendars work like the task manager to implement. To see how this application has been built, go to the section 5.2.

5.2 The missing link: a task application based on CouchDB

The task manager is going to be based on a web user interface, where tasks can be added and removed, and a plugin for Evolution making the management of tasks possible from this program. For the storage, CouchDB is going to be used. This database offers several advantages for the replication process, as it was mentioned on the section 3.4 on page 35. At the moment of writing this thesis CouchDB is not implemented on the iPhone, so the access to the application has to be made through the Safari browser available in the phone. A future improvement could be the development of an iPhone application able to store the data locally. Like this, it would work offline and the synchronization could take place when it comes back online.

In this chapter, the development of the web page is explained, accessing CouchDB through JavaScript. Then, the plugin for Evolution is developed using the `couchdb-glib` and `evolution-couchdb` libraries available to interact with this database. Finally, the replication process is detailed to synchronize both devices (see figure 5.4).

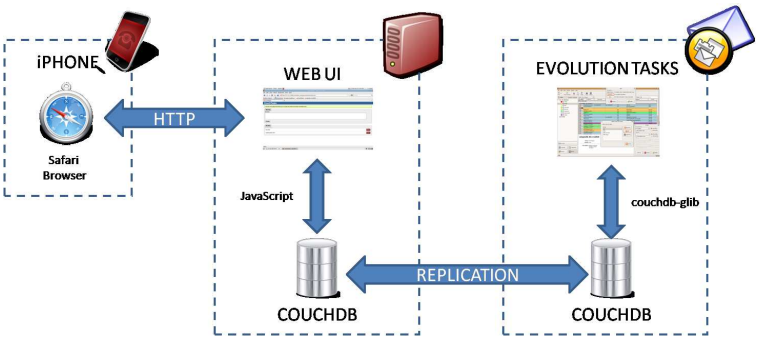

Figure 5.4: Solution architecture

5.2.1 Task Manager Web Application

In this subchapter I will explain how the task manager web application has been built. First, CouchApp is presented – a set of scripts designed to bring clarity and order to the freedom of CouchDB's document-based approach. This tool has been used to create the web application. Second, the main files that compose this application will be analyzed and the functionality of the application will be shown.

5.2.1.1 CouchApp

As I said above, CouchApp is a set of scripts that allow complete, stand-alone CouchDB applications to be built using just HTML and JavaScript.

CouchApp is a Python module and requires Python to be installed in your system. In Linux and Mac OS X Python is usually preinstalled. To check it, you have to introduce the next command in the terminal. If you get an error, Python needs to be installed.

```
$ python -V
```

Next, `python-setuptools` is needed to install CouchApp. Introduce the next command in Ubuntu. If you use another distribution or you don't have a package manager, you will need to build it from the source.

```
$ sudo apt-get install python-setuptools
```

Now, we can use `easy-install` to install CouchDB, SimpleJSON and CouchApp Python modules:

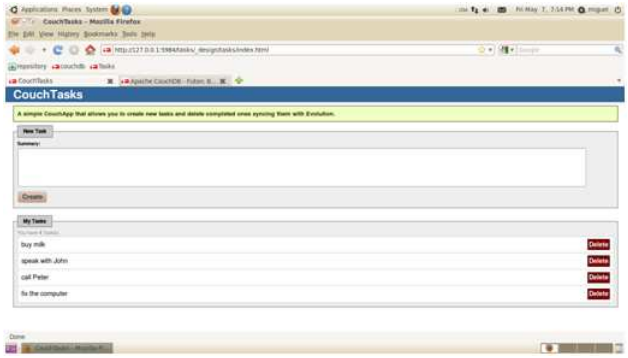

Figure 5.5: Web Application

```
$ sudo easy_install couchdb
$ sudo easy_install simplejson
$ sudo easy_install couchapp
```

CouchApp is installed and you can build CouchDB application in a simple way using only HTML and JavaScript.

5.2.1.2 The task manager web application

The main part of the code shown here is an example of the book "Beginning CouchDB" by Joe Lennon (Chapter 10: Developing CouchDB applications with CouchApp). The code has been modified to make it compatible with Evolution and to adapt it to the iPhone.

This web application allows you to create and delete tasks, displaying any existing tasks when it loads. It has been built using HTML and CSS for the presentation part and JavaScript for the functionality part. The application uses the jQuery JavaScript library as well as some extensions to this library to make the Ajax requests to the CouchDB. The final result is shown in the figure 5.5.

You can find the source code in my repository. To clone it to your local machine, type[2]:

```
$ git clone git://gitorious.org/tasks-evolution-couchdb/couchdb-task-web.git
```

To make this code work, first you have to generate a CouchApp application with the next command:

[2]If you do not have git, install it with

```
$ sudo apt-get install git-core
```

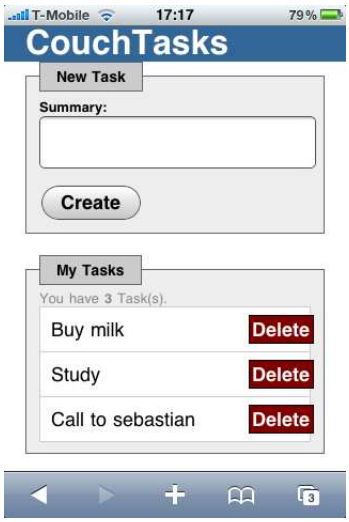

Figure 5.6: Web page version for iPhone

```
$ couchapp generate tasks
```

It creates a new `tasks` database and the default files for the aplication in a new directory called `tasks`. Overwrite the existing files in this directory with the cloned files and be sure to remove the file called `language` in the `tasks` directory. This file causes a fail in the view of the application.

The next command pushs all these documents to the web server offered by CouchApp to make them accessible via a web browser.

```
$ couchapp push . http://127.0.0.1:5984/tasks
```

If you go to this URL, `http://127.0.0.1:5984/tasks/_design/tasks/index.html`, you can access to the web user interface. If this web page is accessed on the iPhone, an adapted version appears, in order to make its usage easier (see figure 5.6). Next, I will explain how this application was developed. Go to the section 5.2.2 on page 55 if you want to skip this part and configure Evolution directly.

To add and delete tasks in CouchDB, a JavaScript file is needed in order to implement the access to the database. This file is placed on the `tasks` directory generated, under the `_attachments/script` subdirectory wit the name `main.js`:

```
$.CouchApp(function(app) {
    $('form#add_task').submit(function(e) {
        e.preventDefault();
        var newTask = {
            summary: $('#summary').val(),
            record_type: "http://www.freedesktop.org/wiki/Specifications/desktopcouch/
                task"
        }
        if(newTask.summary.length > 0) {
            app.db.saveDoc(newTask, { success: function(resp) {
                $('ul#my_tasks').append('<li id="'+newTask._id+'">'
                    +'<div class="summary">'+newTask.summary+'</div>'
                    +'<div class="link">'
                    +'<a href="#" onclick="return false;"'
                    +' id="'+newTask._rev+'">Delete</a>'
                    +'</div>'
                    +'<div class="clear"></clear>'
                    +'</li>');
                $('#'+newTask._rev).click(function() {
                    if(confirm("Are you sure you want to delete this task?")) {
                        var delTask = {
                            _id: newTask._id,
                            _rev: newTask._rev
                        }
                        app.db.removeDoc(delTask, {});
                        $('#'+newTask._id).show().fadeOut(2000);
                        var del_count = parseInt($('#task_count span').html(), 10);
                        del_count--;
                        $('#task_count span').html(del_count);
                        return false;
                    }
                });
                $('ul#my_tasks li:last').hide().fadeIn(1500);
                $('#summary').val('');
                var task_count = parseInt($('#task_count span').html(), 10);
                task_count++;
                $('#task_count span').html(task_count);
            }});
        } else {
            alert('You must enter a description to create a new task!');
        }
    });

    app.view("get_tasks", { success: function(json) {
        json.rows.map(function(row) {
            $('ul#my_tasks').append('<li id="'+row.value._id+'">'
                +'<div class="summary">'+row.key+'</div>'
                +'<div class="link">'
                +'<a href="#" onclick="return false;"'
                +' id="'+row.value._rev+'">Delete</a>'
                +'</div>'
                +'<div class="clear"></clear>'
                +'</li>');
            $('#'+row.value._rev).click(function() {
                if(confirm("Are you sure you want to delete this task?")) {
                    var delTask = {
                        _id: row.value._id,
                        _rev: row.value._rev
                    }
                    app.db.removeDoc(delTask, {});
                    $('#'+row.value._id).show().fadeOut(2000);
                    var del_count = parseInt($('#task_count span').html(), 10);
                    del_count--;
                    $('#task_count span').html(del_count);
                    return false;
                }
            });
        });
        $('#task_count span').html(json.rows.length);
    }});
});
```

With `$.CouchApp(function(app)) {});` I am opening an instance of CouchApp. This code set up the database and design document variables. Within this function, the submit event is captured, avoiding the default option and substituting it for the creation of a new document, where a field summary is set from the value entered for the user in the text area available on the web.

If the user has not entered anything in the text area, an error message is shown. It there are not any errors, the document is saved to the database. If CouchDB reports succeed in this operation, the new task is added on the list showed in

the web page, the text area is cleared and the counter is incremented.

We need a view to be able to retrieve the tasks from the database. If this view does not exist, the application does not maintain the state between sessions. The views are contained under the directory with this name. In this case, the name of the view is `get_tasks`, so a subdirectory with this name is necessary. Within this directory there is a file called `map.js`. It just recovers the summary field of the tasks stored in the database.

Coming back to the `main.js` file within the `_attachments/script` subdirectory, the code part starting with `app.view("get_tasks", {})` loads existing data from CouchDB into the web page using the view described above. To finish, a "Delete" button is added. It lets you delete any task from the database.

This JavaScript file is included in the HTML document (`index.html`) together with the libraries needed for the access to CouchDB.

5.2.2 CouchDB in Evolution Tasks

To add the possibility to store tasks lists in Evolution using CouchDB as database, a plugin needs to be developed. This plug-in will be based on two packages already available for Evolution:

- **Couchdb-glib:** a GLib-based library to allow access to CouchDB databases.
- **Evolution-couchdb:** Evolution backend to access CouchDB databases, used for UbuntuOne integration.

At the moment of writing this thesis the official packages only support contacts. With the next steps, I added the calendar backend and the required methods to make them work with tasks.

First, the installation of these packages, modified to support tasks, is detailed. Then, the changes made are explained and, finally, its functionality is shown. Let's start with the installation.

5.2.2.1 Installation

If you want to check the official GNOME source code (without tasks supporting at the moment of writing this thesis), you can find it in:

```
$ git clone git://git.gnome.org/couchdb-glib
$ git clone git://git.gnome.org/evolution-couchdb
```

Like on the web page, the code used in this chapter can be downloaded from my repository. It is the code of the official packages modified to implement the calendar backend for tasks support. Clone it in your local machine with the next commands:

```
$ git clone git://gitorious.org/tasks-evolution-couchdb/couchdb-glib.git
$ git clone git://gitorious.org/tasks-evolution-couchdb/evolution-couchdb.git
```

Now you should have two directories called `evolution-couchdb` and `couchdb-glib`. To build from the source you need to install some dependencies. For couchdb-glib,

```
$ sudo apt-get install gnome-common gtk-doc-tools libjson-glib-dev
libsoup2.4-dev libsoup-gnome2.4-dev uuid-dev libgnome-keyring-dev
libdbus-glib-1-dev libcurl4-openssl-dev
```

For evolution-couchdb,

```
$ sudo apt-get install libebook1.2-dev libecal1.2-dev libedata-book1.2-dev
libedata-cal1.2-dev evolution-data-server-dev evolution-dev
```

After installing the required dependencies, enter in each directory and build the source with the next commands[3]:

```
$ ./autogen.sh
$ make
$ sudo make install
```

If the installation has been successful, Evolution supports now tasks stored in a CouchDB database. Let's explain how it was made.

5.2.2.2 Development of a task plug-in for Evolution

To add some new functionality to Evolution, EPlugin[4] has to be used, a small and simple system for extending this program. The objective is to show a new configuration window when a tasks list is created in Evolution with the option to use CouchDB as storage database. In this window the needed information for the interaction with the database will be entered and Evolution will be configured to save and recover tasks from CouchDB. The final result is shown in the figure 5.7.

The definition of a plugin for Evolution is made in a XML file. You can find it in the `evolution-couchdb` directory, under the `plugin` subdirectory with the name `org-gnome-evolution-couchdb.eplug.xml.in`.

[3] First, couchdb-glib and, second, evolution-couchdb, because this one requires couchdb-glib to be built.

[4] http://www.go-evolution.org/EPlugin

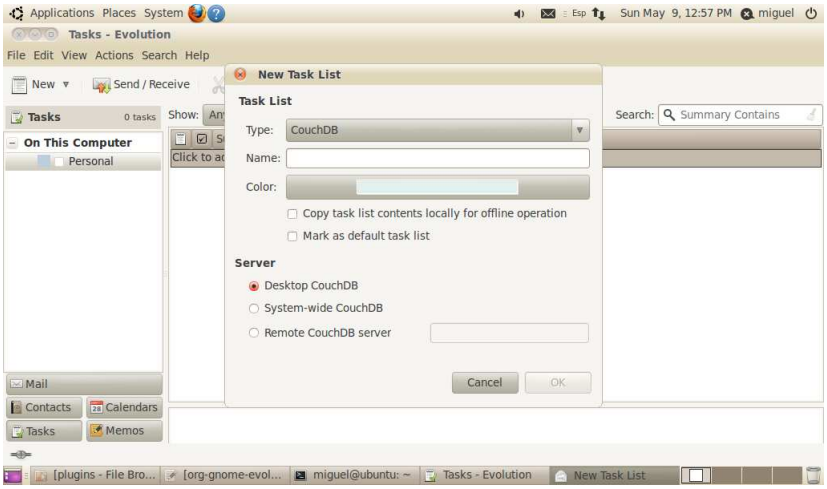

Figure 5.7: Task plugin for Evolution-CouchDB

```
1  <?xml version="1.0"?>
2  <e-plugin-list>
3    <e-plugin id="org.gnome.evolution.couchdb" type="shlib" _name="CouchDB sources"
4      location="@EVOLUTION_PLUGINS_DIR@/liborg-gnome-evolution-couchdb.so" load-on-
           startup="false" localedir = "@LOCALEDIR@" system_plugin="true">
5      <author name="Rodrigo Moya" email="rodrigo.moya@canonical.com"/>
6      <author name="Miguel Rodelas" email="miguel.rodelas@gmail.com"/>
7      <_description>A plugin to setup CouchDB Contacts and Tasks.</_description>
8      <hook class="org.gnome.evolution.addressbook.config:1.0">
9        <group target="source" id="com.novell.evolution.addressbook.config.
             accountEditor" >
10         <item type="item" path="00.general/10.display/00.couchdb" factory="
               plugin_couchdb_contacts"/>
11       </group>
12     </hook>
13     <hook class="org.gnome.evolution.calendar.config:1.0">
14       <group target="source" id="org.gnome.evolution.calendar.calendarProperties" >
15         <item type="item_table" path="00.general/00.source/99.couchdb" factory="
               plugin_couchdb_tasks"/>
16       </group>
17     </hook>
18   </e-plugin>
19 </e-plugin-list>
```

The <e-plugin> element defines an EPlugin. Observe that several of these tags can be packaged in the same XML file under the <e-plugin-list> element. The properties of this plugin are the following ones:

- **id:** A unique string identifying this plugin.
- **type:** The type name of the plugin loader. In this case, "shlib" meaning Shared Library Loader. It requires one extra parameter, location.
- **location:** it contains the full path-name of a shared object to load. liborg-gnome-evolution-couchdb.so in this case.
- **name:** A short name for the plugin.

We can find the following elements inside this tag:

- **author**: The authors' name and mail.
- **description**: A longer description of the plugin's purpose.
- **hook**: A list of all the hooks this plugin wishes to hook into.

For this CouchDB plugin we have two hooks, one for contacts (developed by Rodrigo Moya) and another one for tasks (developed by me). Let's study the tasks one, the added for the purpose of this thesis. It is a configuration page hook which allows us to enter the required configuration information in Evolution for the interaction with the CouchDB database.

The <group> element represents a group of configuration items and it is composed of two elements:

- **id**: The name of the configuration window to which this hook applies.
- **target**: The type of target this configuration window applies to. This will normally be tied directly to the specific configuration window itself.

The properties for the <item> element are as followed:

- **type**: The menu item type.
- **path**: The path to the configuration item in question. It will place the item in the right order.
- **factory**: the factory method used to create the GtkWidget elements for this configuration item. In our case, plugin_couchdb_tasks. This method will create the configuration windows as we want to build it. You can find the implementation of this method in the same `plugin` subdirectory under the file named `couchdb-tasks-source.c` together with other required methods for the plugin.

Now that the plugin has been built and the configuration window is shown when CouchDB is chosen like new tasks list type, let's implement the calendar backend. For that, first Evolution Data Server needs to be briefly presented.

EVOLUTION DATA SERVER

Evolution Data Server (EDS) manages the access to the address book, calendar and tasks information available for Evolution and other applications. It is a CORBA element which allows concurrent access by several client applications to the same data together with notifications of changes. It can be extended with the addition of plugins to manage different kinds of address books/calendars/tasks sources by writing a shared library, which will be loaded by EDS on startup.

Calendar and tasks share the two libraries needed by the client to access the data:

- **libecal**: It is the main client library for calendar and tasks, implementing all the necessary to open and manage this kind of sources.
- **libical**: It parses iCalendar text and offers a set of tools for dealing with iCalendar.

Our purpose is to add a new kind of source for tasks, CouchDB. So we have to write a shared library and load it when EDS starts. To do this, a new calendar backend need to be written.

The calendar backend deals with the communication between EDS and the specific calendar servers/types. There are backends for Groupwise, Exchange, Webcal, ... The CouchDB backend has to traduce basically the operations make in Evolution (create a new tasks list, new task, delete task, ...) to the operations needed in the CouchDB database (create a new database, create a new document, delete a document, fill in a document with the data required, ...) and vice versa.

To load a new backend in EDS, the shared library has to be installed in a known place[5]. When EDS loads the library, it looks for some named symbols:

- **eds_module_initialize**: register all ECalBackendFactory-derived classes. These factories are responsible for creating backends of a given kind (events, tasks and journal), so a factory class for each kind must be registered. For each ECalBackendFactory-derived classes, three methods need to be implemented:
  - get_kind: returns the kind of the backends created by the factory.
  - get_protocol: returns the protocol used by the backends.
  - new_backend: creates a new backend for the given kind.
- **eds_module_shutdown**: It cleans up all the memory used by the extension.
- **eds_module_list_types**: A list of the factory types implemented by the extension.

Extensions need to implement these methods for being loaded when EDS starts. You can find the implementation of these methods for the CouchDB backend in the `evolution-couchdb` directory, under the `calendar` subdirectory with the name `e-cal-backend-couchdb-factory.[ch]`.

To complete the backend, the methods required to translate the API used by EDS calendar clients into CouchDB operations need to be implemented. This is made by writing a subclass of `ECalBackend` (for asynchronous mode). If synchronous mode would be required, the class to implement would be `ECalBackendSync`.

[5] `$evolution_prefix/lib/evolution-data-server-$VERSION/extensions`

ECalBackend contains the following virtual methods, which need to be implemented by the CouchDB backend. You can find again the implementation for this thesis under `evolution-couchdb/calendar` with the name `e-cal-backend-couchdb.c`.

- **is_read_only**: returns whether the calendar is read only or not.
- **get_cal_address**: returns the email address of the owner of the calendar.
- **get_alarm_email_address**: returns the email address to be used for alarms.
- **get_ldap_attribute**: returns specific LDAP attributes.
- **get_static_capabilities**: returns the capabilities provided by the backend, like whether it supports recurrences or not, for instance.
- **open**: opens the calendar/tasks list.
- **remove**: removes the calendar/tasks list.
- **create_object**: creates a new event/task in the calendar/task list.
- **modify_object**: modifies an existing event/task.
- **remove_object**: removes an object from the calendar/task list.
- **discard_alarm**: discards an alarm (removes it or marks it as already displayed to the user).
- **receive_objects**: import a set of events/tasks in one go.
- **send_objects**: send a set of meetings in one go, which means, for backends that do support it, sending information about the meeting to all attendees.
- **get_default_object:** returns an empty object with the default values used for the backend.
- **get_object**: returns an event/task, given its UID.
- **get_object_list**: returns a list of events/tasks given a set of conditions.
- **get_timezone**: returns time zone objects for a given TZID.
- **add_timezone**: adds a time zone to the backend.
- **set_default_timezone**: sets the time zone to be used as the default.
- **start_query**: starts a live query on the backend.
- **get_mode**: returns the current online/offline mode for the backend.

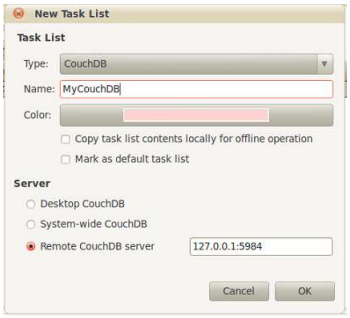

Figure 5.8: Configuration window for a new task list

- **set_mode**: sets the current online/offline mode.
- **get_free_busy**: returns Free/Busy information for a list of users.
- **get_changes**: returns a list of changes made since last check.

There are also a couple of internal methods, not used by clients, but by the `ECalBackend` class itself, which are:

- **internal_get_default_timezone**: returns the default time zone.
- **internal_get_timezone**: returns a given time zone.

Not every one of these methods should be implemented. The basic funcionality is achieved with `open`, `remove`, `create_object`, `remove_object` and `start_query`. See the file mentioned above for more details.

5.2.2.3 Evolution configuration to manage CouchDB task lists

First, open Evolution and go to the Task section. Click on `New→New Task List`. The window in the figure 5.8 appears. The configuration fields are:

- **Type**: "CouchDB"
- **Name**: The name of your task list
- **Server**: The kind of server to be used
  - **Desktop CouchDB**: this option is for a future integration with UbuntuOne, a storage application and service to enable users to store and sync files and personal information between computers. At present it is working with contacts, notes and files already. Tasks will be added when the integration with Evolution is completed.

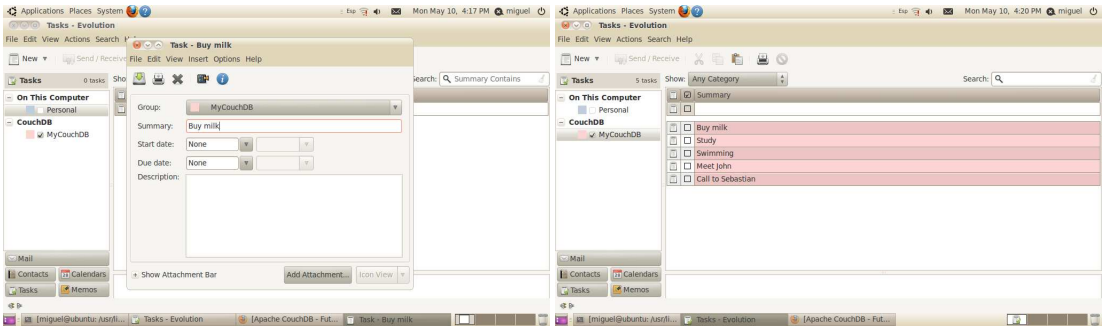

Figure 5.9: New task and task list windows

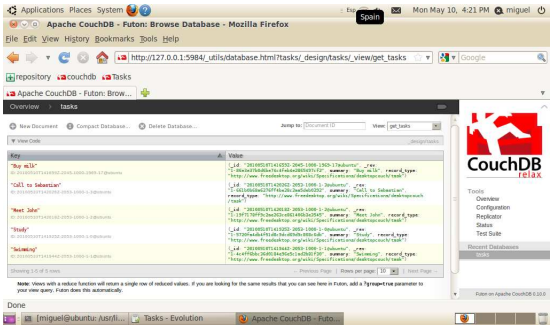

Figure 5.10: CouchDB view for the task list

- **System-wide CouchDB**
- **Remote CouchDB server**: Introduce the IP address and the port of your CouchDB database installation. If this is in your local machine, type here `127.0.0.1:5984`, being 5984 the port by default.

A new task list appears on the left. If you add a new task through the `New→Task` option, the window 5.9 is shown, where you can enter the details for this task[6]. In the same figure you can appreciate the look of a list with several tasks.

If we look at CouchDB using the same view created for the web page, the result is shown in the figure 5.10.

So now it is possible to manage your task list from any place with a browser available (personal computers, mobile phones, laptops, ...) and you can have it synced with Evolution.

[6] At the moment of writing this thesis, only the summary field is implemented. Tasks can be added, removed and queried. The rest of fields and more funcionalities will be added in following versions.

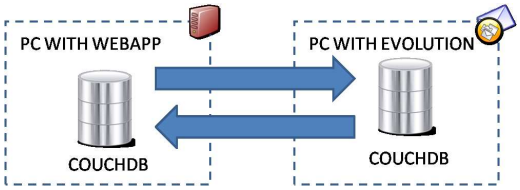

Figure 5.11: Replication scenario

Figure 5.12: Replication configuration

5.2.3 Replication

The task manager application has been developed. It is composed of a web user interface and a plug-in for Evolution. The question now is how to synchronize tasks in the scenario required, where Evolution is on a computer and the web application is installed in other device[7], each one with its own database and without a permanent connection between them (see figure 5.11). When they come back online, they should make a replication.

The replication process can be automated in the code if it is desired. This chapter explains how to replicate using Futon, the web user interface of CouchDB.

You can access Futon in `http://127.0.0.1:5984/_utils`. Check in the `Configuration` section, under the `httpd` options if `bind_address` is set to `0.0.0.0`. By default the CouchDB installation only accepts petitions in local. With this value of the parameter, any other machine can access to the CouchDB database, necessary for the replication process. These parameters are loaded from the file `local.ini` (in `/usr/etc/couchdb` in a Ubuntu installation by default) but the easiest way is to change it with Futon.

Go to the `Replicator` section. You will find the configuration fields shown in the figure 5.12. As you can observe, the replication is only one-way. The two-way replication can be easily achieve making the process in both directions. Just choose `tasks` as local database and enter `http://[IP device]/tasks` as remote database and make the replication in both ways. Tasks existing in each one of the local databases have to be now in both CouchDB instances.

The task application is in a start state at the moment, so conflicts are not possible in the way explained in this thesis because these items are composed only of one field, summary. However, when the development is complete, there

[7] Remember that the iPhone cannot access directly to a CouchDB database. It uses the web application, residing in a server, to manage the tasks.

could be conflicts updating elements with several fields. In this situation, the application needs to manage these conflicts in order to implement the "merging" and "newer wins" policies. CouchDB only chooses one of the versions as winner with a deterministic algorithm in every instance of the database, remaining the loser versions on it until a compaction operation. The user will lose data in a scenario with merging as possibility if nothing is done, the PIM application is the responsible to merge the elements when possible or apply the "newer wins" policy in other case.

Chapter 6

Conclusions

This thesis has put in evidence several failures in the existing synchronization systems between the iPhone and Evolution. Most of these problems come from the use of a closed operating mobile system, the iPhone OS, with no many chances to open source solutions. Moreover, the desktop application to synchronize belongs to the Linux system, while almost every commercial solution has been thought to work on Windows. So the synchronization of PIM data between these devices has meant a big challenge.

First, any device includes an UID and time stamp fields. That makes almost impossible a synchronization system without unnecessary replications and lose of data. Together with this, any of the tested servers are able to manage a "newer wins" policy and merging is not well implemented in the majority of them.

Several of the involved protocols only work when a connection is available, as GroupDAV, LDAP, WebDAV with Toodledo or the Evolution plugin for Google. That does not fulfill the basis requirements of a synchronization system, which essence is to let users work offline making the data consistent when coming back online.

In order to fix all these lacks, a new synchronization system based on CouchDB is proposed. Taking the task application developed in this thesis as a basis, the same could be made with contacts, calendars and memos (see figure 6.1). In this moment, the UbuntuOne project is oriented in this way. It supports contacts already and tasks will be added as soon as the implementation is completed. Moreover, our own server can be used with this project, not being necessary to store our data in Ubuntu servers.

Another improvement for this system would be the development of an iPhone application able to store data locally in order to let us work when a connection is not available. The synchronization with the CouchDB would be made when coming back online, achieving a complete synchronization system.

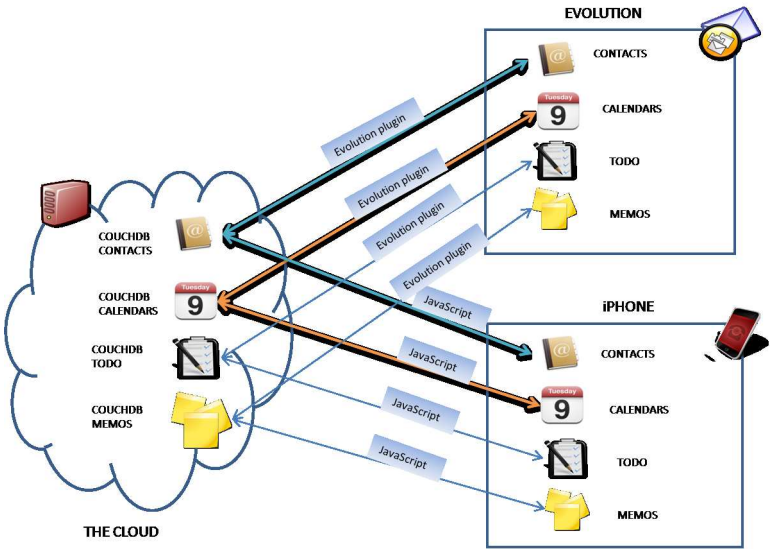

Figure 6.1: Proposed Solution based on CouchDB completely

That is the basis of a new PIM synchronization system with the conflict detection and resolution in an optimal way as main challenge to achieve. If you are interested, you can follow the new updates for the application in the repositories mentioned in the thesis.

Bibliography

[1] On the Scalability of Data Synchronization Protocols for PDAs and Mobile Devices. (S.Agarwal D. Starobinski A. Trachtenberg).
(http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.12.5572&rep=rep1&type=pdf)

[2] Developing and Managing Mobile Applications with SyncML and Funambol.
(http://www.funambol.com/documents/Funambol_SyncML_Book.pdf)

[3] Designing Mobile Applications: Why Sync Is Central.
(http://www.sybase.com/files/White_Papers/ias_wp_why_sync_is_central.pdf)

[4] Why sync is so difficult? By Jean-Gabriel Morard.
(http://gigaom.com/2009/05/10/why-sync-is-so-difficult/)

[5] PIM Data Synchronization: Why is it so hard? Contributed by Patrick Ohly.
(http://lwn.net/Articles/333441/)

[6] SyncML, Wikipedia
(http://en.wikipedia.org/wiki/SyncML)

[7] Developing and Managing Mobile Applications with SyncML and Funambol
(http://www.funambol.com/documents/Funambol_SyncML_Book.pdf)

[8] SyncML – "Building an Industry-Wide Mobile Data Synchronization Protocol", White Paper, 2000
(http://xml.coverpages.org/SyncML-WhitePaper.pdf)

[9] SyncML – "SyncML Sync Protocol, version 1.1", 2002
(http://www.openmobilealliance.org/tech/affiliates/syncml/syncml_sync_protocol_v11_20020215.pdf)

[10] SyncML – "SyncML Representation Protocol, version 1.1", 2002
(http://www.openmobilealliance.org/tech/affiliates/syncml/syncml_represent_v11_20020215.pdf)

[11] Wikipedia. WebDAV
(http://en.wikipedia.org/wiki/Webdav)

[12] Beginning CouchDB, by Joe Lennon

[13] CouchDB Official Website
(http://couchdb.apache.org/)

[14] EPlugin
(http://www.go-evolution.org/EPlugin)

[15] Evolution Data Server Architecture
(http://www.go-evolution.org/EDS_Architecture)

[16] Evolution. Developers information.
(http://projects.gnome.org/evolution/developer.shtml)